COL728 Minor1 Exam
Compiler Design
Sem II, 2016-17

Answer all 5 questions Max. Marks: 20

1. Short questions

a. Show that every regular language is also a context-free language [2]

We know that every regular language can be represented by a regular expression which can have terminal symbols and union, concatenation and kleene operators. If we can show that all regular expressions can be written in form of CFG grammer rules which have only a non terminal on LHS, then we can say that every regular language is also a CF Language.
So if a regular expression has only a terminal symbol a, then it can be represented by grammar rule S -> a
Any concatenation between two regular expressions S1 and S2 (S1.S2) can be represented by grammar rule S -> S1.S2
Any union between two regular expressions S1 and S2 (S1+S2) can be represented by grammar rule S -> S1 | S2
And kleene star on a regular expression S (S*) can be represented by grammar rule S -> S.S | ε

Thus we can represent any regular expression in form of CFG rules. So we can say that every regular language is also a context-free language.

[Full marks for translating regular grammar or DFA to Contect free grammar.]

b. Why does the lexer use regular languages but the parser use context-free languages? Explain both reasons briefly and clearly, i.e., (1) why does the lexer use regular languages, and (2) why does the parser use context-free languages. [2]

(1) Typically we want to design the language for the tokens/identifiers so that it is easy to identify them. Regular languages is perhaps the simplest class of languages that are easy to identify using a deterministic finite automaton.

(2) Certain intuitive programming patterns, like balanced parentheses, are commoly required in programming languages. These constructs are not expressible through regular languages. Hence, typically the language of valid programs is based on a more general class of languages, namely context-free languages.

[1 Mark for each reason, reason should be something similar to this. It is important to say that regular languages are the simplest class of languages and hence most programming languages rely on it. It is also important to say that regular languages do not capture certain common and intuitive programming patterns.]

2. Consider the following grammar:

E → if E then E
E → if E then E else E
E → id

a. Show that this grammar is ambiguous [2]

Suppose we have a string “if id then if id then id else id” and we want to make its left most derivation tree. Then there are at least two LMD trees for it as shown below:

E -> if E then E	if E then E
E -> id	if id then E
E -> if E then E else E	if id then if E then E else E
E -> id	if id then if id then E else E
E -> id	if id then if id then id else E
E -> id	if id then if id then id else id

Another way

E -> if E then E else E	if E then E else E
E -> id	if id then E else E
E -> if E then E	if id then if E then E else E
E -> id	if id then if id then E else E
E -> id	if id then if id then id else E
E -> id	if id then if id then id else id

Hence there are multiple LMDs for the same string in this grammar, so this grammar is ambiguous. In other words, the grammar is ambiguous because it has two parse trees for some string.

[full marks for correct example; 1 mark if only meaning of ambiguity is well understood]

b. Write an alternative grammar that accepts the same language but is not ambiguous. Clearly state the rule that you used to resolve the ambiguity (any rule that resolves ambiguity is okay). [3]

Unambiguous grammar accepting the same language is written below. Ambiguity in above grammar was because else could be assigned to any if instead of the nearest unmatched if that came before it. So we created two different non-terminals: one representing the set of balanced strings and another representing the set of unbalanced strings. A string is balanced if every if-then is also matched with an else. A string is unbalanced if some if-then is not matched with an else.

An “else” can appear only if the string inside the “then” clause is balanced (because otherwise the “else” should have matched the unblanced clause).

E -> E_balanced | E_unbalanced
E_balanced -> if E then E_balanced else E_balanced | id
E_unbalanced -> if E then **E_balanced** else E_unbalanced | if E then E

[If any grammar is stated that removes ambiguity of to which “if”, an “else” belongs, and covers all strings then it gets full marks; missing some strings looses 1 mark, still ambigous looses 2 marks]

3. Consider the following grammar:

E → id
E → (E)
E → E.E

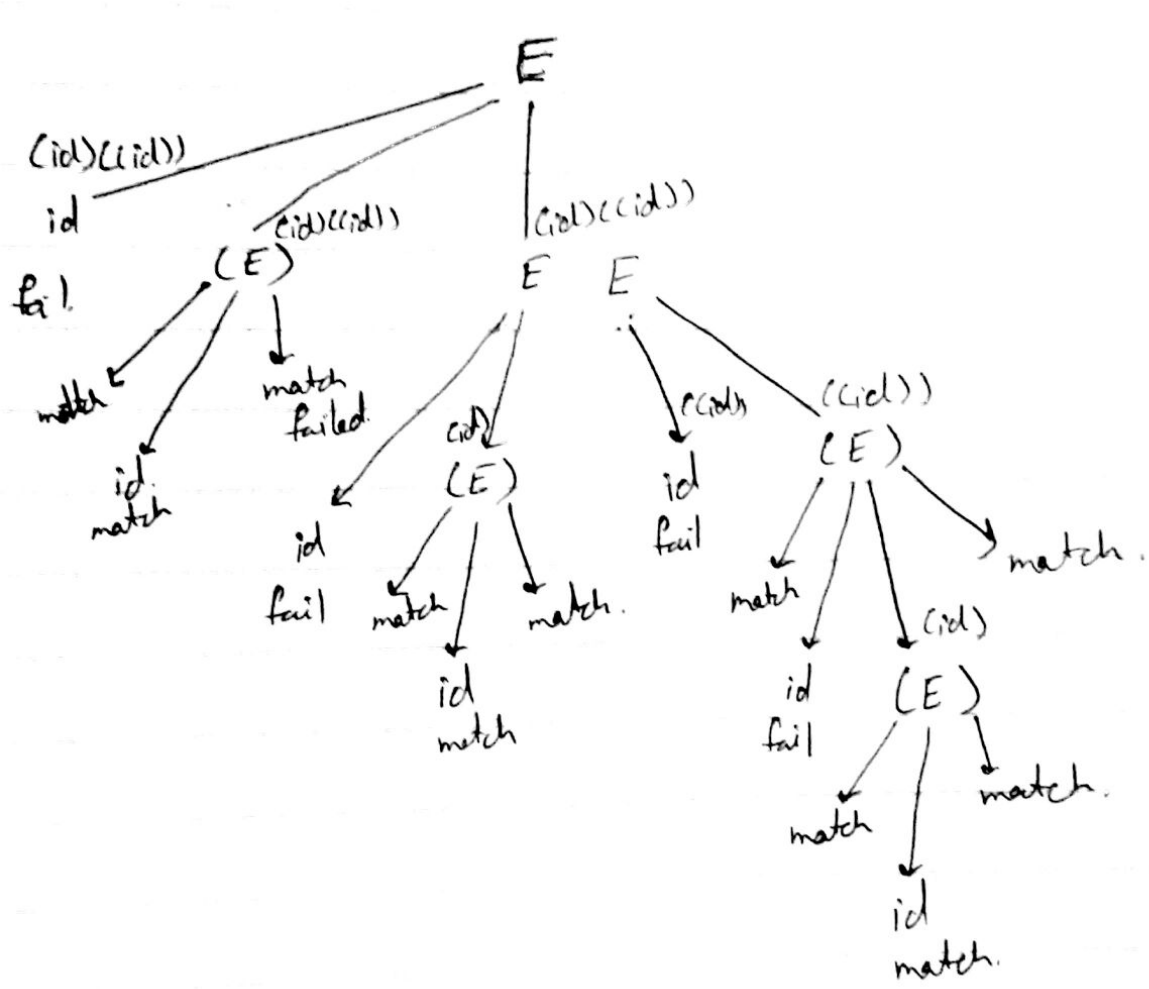
Run the recursive descent parsing algorithm (with backtracking) on the following input:

(id)((id))

Clearly and concisely show the steps and the final result. No marks for partial or incorrect execution [4]

To start with, we should add a new start symbol which includes the $ character to signify that it matches only if the end of the input has been reached:

S → E$
Can draw a recursion tree as follows (shown without S here).



The answer should clearly point out that the match fails for “(id)” because the end of input has not been reached. Ideally the student should have used the S → E$ production to capture this.

4. Consider the following grammar, and answer the following questions clearly.

E → id
E → (E * E)
E → (E + E)

a. Is this grammar LL(1)? [2]

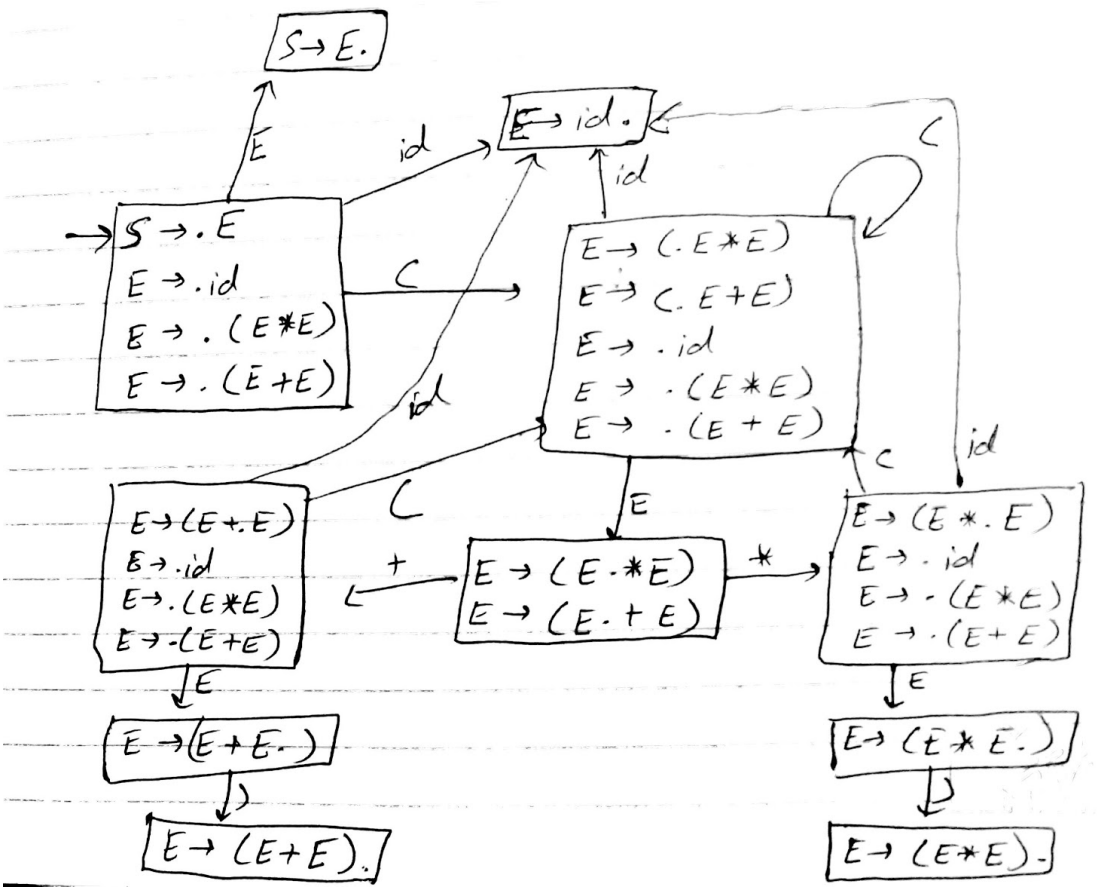
No the grammer is not LL(1) as there are two rules for same non terminal E which start with "(", so grammar is not left factored and hence is not LL(1).

We can also say that First ("(E+E)") = "(" and First("(E*E)") = "("
So in parsing table, T[E, (] will have both (E+E) and (E*E) causing a conflict. So the grammer is not LL(1).

[full marks for correct reason; No marks for just yes or no]

b. Is this grammar SLR? [4]

DFA of items for SLR parsing of this grammar is shown below. There are no Shift reduce or reduce reduce conflicts in them, so this is SLR grammar.



[full marks for correct reason; No marks for just yes or no; partial marks for partially correct DFA]

5. Give an example of a program (in a programming language of your choice) that is a correct program as far as the parser is concerned (i.e., the parser accepts that program as a valid program), but is not a correct program otherwise (i.e., the semantic analysis step would discard the program as invalid). [1]

Many examples. E.g., a C program with addition operator between a function identifier and an integer etc. Basically any program in which we expect the type checker to fail.

[Full marks for correct example]

Alternate working for Q3:

Functions for this grammer are:
Bool token(x) {

```
    return (next++ == x);
}
Bool S() {
    Return E() && token('$');
}
Bool E() {
    TOKEN *save = next;
     Return ((next=save, E1()) || (next=save, E2()) || (next=save, E3()))
}
Bool E1() {
    return token('id');
}
Bool E2() {
    Return (token('(') && E() && token(')'));
}
Bool E3() {
    Return (E() && E());
}

Input is
Tokens : (   id   )   (   (   id   )   )   END
Indexes: 0  1    2  3   4  5  6   7   8

Execution sequence will be:

call S
call E
  call E1
          terminal id != (          false

  ||

  call E2
          terminal ( == (                   true
          &&
          call E
          call E1
          terminal id == id         true
          &&
          terminal ) == )                   true
          fail as next is not equal  to End of Input

  ||

  call E3
          call E
```

```
        call E1
        terminal id != (        false

        ||

        call E2
        terminal ( == (         true
        &&
        call E
        call E1
        terminal id == id       true
        &&
        terminal ) == )         true
        &&
        call E
        call E1
        terminal id != (        false

        ||

        call E2
        terminal ( == (         true
        &&
        call E
        call E1
        terminal id != (        false

        ||

        call E2
        terminal ( == (         true
        &&
        call E
        call E1
                terminal id == id   true
        &&
        terminal ) == )         true
 true
```

Note :- Although this sequence will never be able to parse (id id). So it seems wrong somewhere.

So final result is parse is successful with LMD :-

S -> E$ -> EE$ -> (E)E$ -> (id)E$ ->(id)(E)$ -> (id)((E))$ -> (id)((id))$